Chapter 6

Flow of Control Part 2:

Looping

Topics

- Event-Controlled Loops Using *while*
- Looping Techniques
- Type-Safe Input Using *Scanner*
- Constructing Loop Conditions
- Testing Techniques for *while* Loops
- Event-Controlled Loops Using *do/while*
- Count-Controlled Loops Using *for*
- Nested Loops

HOME

The Grocery Cashier

- A grocery cashier's job is to calculate the total costs of the items in the cart.
  - The cashier starts with a total of $0.00.
  - The cashier scans an item to get its price and adds the price to the total.
  - The cashier scans the next item to get its price and adds the price to the total.
  - …
  - When there are no more items to scan, the total is complete.
- *Notice that the cashier is performing the same operations on each item!*

HOME

Looping

- In computing, we often need to perform the same operations on multiple items.
- Typically, these tasks follow this pattern:
  - initialize values (set total to 0)
  - process items one at a time (add price to total)
  - report results (report total)

The flow of control that programmers use to complete jobs with this pattern is called **looping**, or **repetition**.

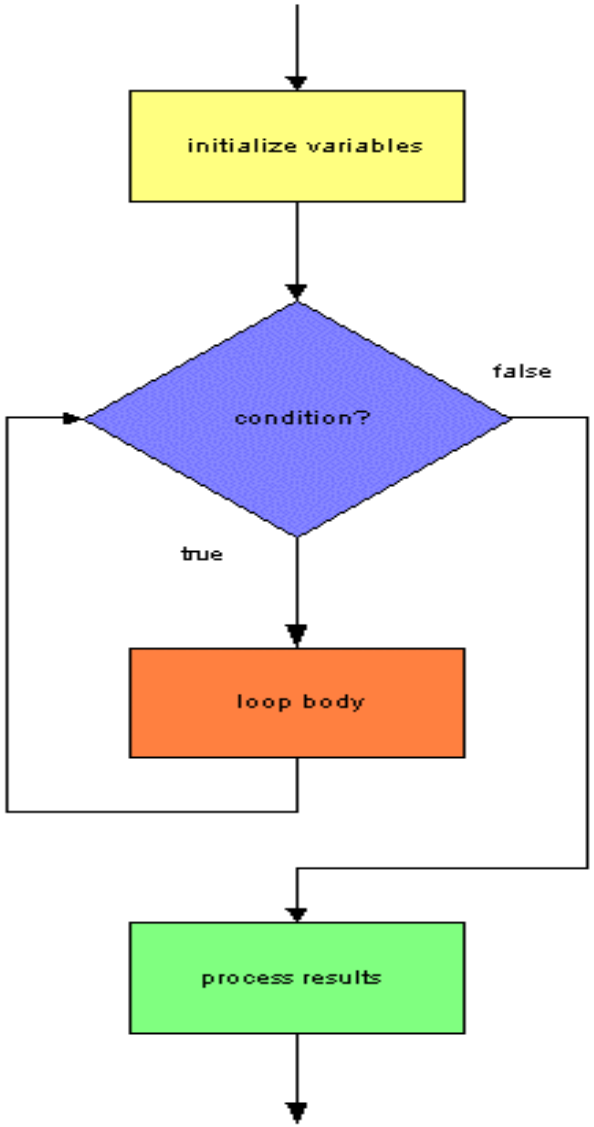
HOME

The *while* Loop

- The *while* loop is designed for repeating a set of operations on data items when we don't know how many data items there will be.
- We will get some signal when we have reached the end of the items to process. (For the grocery cashier, it's the divider bar)
- The end of data items could be indicated by a special input value called a **sentinel value** or by reaching the **end of a file**
- Receiving the signal is an **event**; we call this **event-controlled looping**

HOME

while Loop Flow of Control



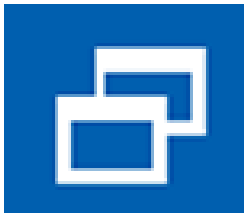
HOME

while Loop Syntax

```
//initialize variables
while ( boolean expression )
{
  // process data (loop body)
}
//process the results
```

**Note: curly braces are optional if only one statement is in the loop body

HOME



- Indent the body of a *while loop* to clearly illustrate the logic of the program.

HOME

Operation of the *while* Loop

- If the condition evaluates to *true*, the loop body is executed, then the condition is re-evaluated.
- As long as the condition evaluates to *true*, we continue to repeat the loop body.
- The loop body must "update the loop condition"; that is, it must perform some operation that eventually will cause the loop condition to evaluate to *false*
- Typically, the loop update will be an attempt to read the next input value, in order to detect the sentinel value or the end of the file.

HOME

Some Definitions

- **iteration**
  - one execution of the loop body
- **loop update**
  - One or more statements that could cause the loop condition to evaluate to *false* (to end the looping)
- **loop termination condition**
  - the event that causes the loop condition to evaluate to *false*

HOME

The Endless Loop

- also called an **infinite loop**
- If the loop condition never evaluates to *false*, the loop body is executed continuously, without end
- If the loop body has no output, the endless loop makes the computer appear to hang.
- If the loop body produces output, the endless loop results in that output being repeatedly written without end.
- Aborting the program will interrupt the endless loop.

HOME

Pseudocode for the Grocery Cashier

```
set total to $0.00
reach for first item
while item is not the divider bar
{
   get price of item
   add price to total
   reach for next item  // loop update
}
// if we get here, the item is the
// divider bar
output the total price
```

HOME

- Avoid putting a semicolon after the condition of a *while* loop. Doing so creates an empty loop body and could result in an endless loop.
- This code causes an endless loop:

```
int i = 0;
while ( i < 10 );  // empty loop body
{
  i++;  // not in the loop body
}
```

- The semicolon indicates an empty loop body; *i++* is never executed because it is not part of the loop body, so the condition is always *true*.

HOME

Sentinel-Controlled *while* Loop

```
initialize variables
// priming read
read the first data item
while ( item is not the sentinel value )
{
    process the item

    // update read
    read the next data item
}
report the results
```

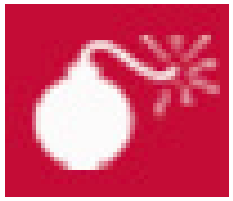
HOME

- Omitting the update read may result in an endless loop.
- Example:

```
System.out.print( "Enter a value > " );
int input = scan.nextInt( );
while ( input != 10 )  // 10 is sentinel value
{
  System.out.println( input );
}
```

- If the value entered for *input* is not *10*, this is an endless loop because we never read a new value for *input*. Thus, the condition always evaluates to *true*.

HOME



- Omitting the priming read can lead to incorrect results.
- Example:

```
int input, count = 0;;
while ( input != 10 ) // 10 is sentinel value
{
  System.out.print( "Enter an integer > " );
  input = scan.nextInt( );
  count++;
}
System.out.println( "Count is " + count );
```

- If the user enters the values *20 30 10*, then the output will be *"Count is 3"*, which is incorrect. We should not process the sentinel value.

HOME

Example 6.1

- *EchoUserInput.java*
- -1 is the sentinel value
- We read integers from the user until the user enters -1
- To process the data, we echo the user input to console

HOME

Reading from a Text File

```
initialize variables
while ( there is more data in the
  file )
{
    read the next data item
    process the data
}
report the results
```

HOME

Setup for Reading from a File

- *File* class (*java.io* package) constructor

```
File ( String pathname )
```
constructs a *File* object with the file name *pathname*

- A *Scanner* constructor for reading from a file

```
Scanner( File file )
```
creates a *Scanner* object associated with a file

Example:

```
File inputFile = new File( "input.txt" );
Scanner scan = new Scanner( inputFile );
```

HOME

Scanner Class *hasNext* Method

- Use this method to detect the end of the input values

Return type	Method name and argument list
boolean	`hasNext( )` returns *true* if there is more data to read; returns *false* when the end of the file is reached

Eliminates the need for a priming read because the *hasNext* method looks ahead for input.

- An *IOException* may be generated if we encounter problems reading the file. Java requires us to acknowledge that these exceptions may be generated. One way to do this is to add this clause to the *main* definition

```
throws IOException
```

- *See Example 6.2 reading from a text file*

HOME

Looping Techniques

- There are standard patterns and techniques for performing these common operations:
  - Accumulation
  - Counting Items
  - Finding an Average
  - Finding Maximum or Minimum Values
  - Animation

HOME

Accumulation

- Approach: the running total
  - We start by initializing a total variable to 0.
  - Each time we read a value, we add it to the total.
  - When we have no more values to read, the total is complete.

- Note that this is the same pattern used by the grocery cashier.

HOME

Accumulation Pseudocode

```
set total to 0    // very important!
read a number    // priming read
while ( number is not the sentinel value )
{
    add the number to total

    read the next number // update read
}
output the total
```

- *See Example 6.3 Calculator.java*

HOME

- Forgetting to initialize the total variable to 0 before beginning the loop will produce incorrect results.

HOME

Counting Items

- Approach: the running count
  - We start by initializing a count variable to 0.
  - Each time we read a value, we check whether that value meets the criteria as something we want to count. If so, we increment the count variable by 1.
  - When we are finishing reading values, the count is complete.

HOME

Counting Items Pseudocode

```
set count to 0    // very important!!
read input        // priming read
while ( input is not the sentinel value )
{
    if ( input is what we want to count )
         add 1 to count

    read the next input  // update read
}
output count
```

- *See Example 6.4 CountTestScores.java*

HOME

- Forgetting to initialize the count variable to 0 before beginning the loop will produce incorrect results.

HOME

Calculating an Average

- Approach: combine accumulation and counting
- We start by initializing a total variable and count variable to 0.
- Each time we read an item, we add its value to the total variable and increment the count variable
- When we have no more items to read, we calculate the average by dividing the total by the count of items.

HOME

Calculating an Average Pseudocode

```
set total to 0
set count to 0
read a number
while ( number is not the sentinel value )
{
   add the number to total
   add 1 to the count

   read the next number
}
set average to total / count
output the average
```

- *See Example 6.5 AverageTestScore.java*

HOME

- Forgetting to check whether the denominator is 0 before performing division is a logic error.
- In integer division, if the divisor is 0, an *ArithmeticException* is generated.
- In floating-point division, if the divisor is 0:
  - If the dividend is also 0,
    - the result is *NaN*
  - If the dividend is not 0,
    - the result is *Infinity*

HOME

Correct Calculation

- Remember that if we declare *total* and *count* as integers, then *average* will be calculated using integer division, which truncates the remainder.
- To get a floating-point average, we need to type cast one of the variables (either *total* or *count*) to a *double* or a *float* to force the division to be performed as floating point.
- Example:

```
double average = (double) ( total ) / count;
```

HOME

Finding Maximum/Minimum Values

- Approach: the running maximum or minimum
- For the maximum (minimum is similar):
  - Read the first item and save its value as the current maximum
  - Each time we read a new value, we compare it to the current maximum.
    - If the new value is greater than the current maximum, we replace the current maximum with the new value.
  - When we have no more items to read, the current maximum is the maximum for all values.

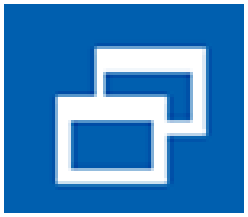
HOME

Finding Maximum Value
Pseudocode for Reading From a File

```
read a number
make that number the maximum
while ( there is another number to read )
{
   read the next number
   if ( number > maximum )
   {
      set maximum to number
   }
}
output the maximum
```

- ***See Example 6.6***

HOME



- Initializing a maximum or a minimum to an arbitrary value, such as 0 or 100, is a logic error and could result in incorrect results.

- For example, if we initialize the maximum to 0 and all the values read are less than 0, then we will incorrectly report 0 as the maximum.
- Similarly, if we initialize the minimum to 0 and all the values read are greater than 0, then we will incorrectly report 0 as the minimum.

HOME

Animation

- Move object (for example, a ball) across window by changing *x* and *y* values and redrawing
- Drawing a figure using offsets from (x, y) makes this possible.
- Loop terminates when we reach the right edge of window

```
figure + width >= windowWidth
```

- Thus, our loop condition becomes:

```
figure + width < windowWidth
```

HOME

Circle Class API

- Constructor

```
Circle( int sX, int sY, int sDiam,
            Color sColor )
```

constructs a *Circle* object; sets *x* and *y* to *sX* and *sY*, respectively; *diameter* to *sDiam*; and *color* to *sColor*.

- Example:

```
Circle c = new Circle( 100, 200, 50,
                       Color.RED );
```

instantiates a ***Circle*** object with the upper left (x,y) coordinate of (100, 200), a diameter of 50, and the color red.

HOME

Circle Class Methods

Return type	Method name and argument list
`int`	`getX( )` returns the *Circle*'s x value
`void`	`setX( int newX )` sets the *Circle*'s x value to *newX*
`int`	`getY( )` returns the *Circle*'s y value
`void`	`setY( int newY )` sets the *Circle*'s y value to *newY*
`int`	`getDiameter( )` returns the *Circle*'s diameter
`void`	`setDiameter( int newDiameter )` sets the *Circle*'s diameter to newDiameter

HOME

Circle Class Methods (con't)

Return type	Method name and argument list
`Color`	`getColor( )` returns the *Circle*'s color
`void`	`setColor( Color newColor )` sets the *Circle*'s color to *newColor*
`void`	`draw( Graphics g )` draws a filled circle with x and y being the upper-left corner of a bounding rectangle, and with the diameter and color set in the object.

HOME

Animation First Attempt

- *See **Example 6.7 RollABall1.java***
- Get width of window by calling the *getWidth* method in the *JApplet* class with this API:

Return type	Method name and argument list
`int`	**`getWidth( )`** returns the current width of the window in pixels

- In *while* loop, we increment *x* by the ball diameter, plus a constant space between balls (*SPACER*)

```
ball.setX( ball.getX( ) +
           ballDiameter + SPACER );
```

HOME

Result

- All balls appear at once
- Solution:
  - Slow down the time between each drawing of the ball.
  - Erase the current ball before drawing the next one

HOME

Slow Down the Execution

- We can slow down the animation by calling the *static wait* method in the author's *Pause* class from the *while* loop body

Return type	Method name and argument list
`void`	**`wait( double seconds )`** *static* method that pauses execution for the number of *seconds* specified.

- Example:

```
Pause.wait( .03 );  // pause 3/100th of a second
```

HOME

Erase the Ball

- To erase the previous ball before drawing the new ball
- Call *clearRect* method in *Graphics* class:

Return type	Method name and argument list
`void`	**`clearRect`**`( int x, int y, int width, int height )` draws a filled rectangle in the background color

- Get window height by calling *getHeight* method in *JApplet* class

Return type	Method name and argument list
`int`	**`getHeight`**`( )` returns the current height of the window in pixels

HOME

Pseudocode for Animation

```
set starting (x, y) coordinate
instantiate the ball object
while ( x + diameter is within the window )
{
     draw the ball
     pause
     erase the ball
     set (x, y) coordinate to next
               drawing position
}
```

- *See **Example 6.8 RollABall2.java***

HOME

Input Problems

- What happens if the user does not enter the data type we request?
  - The *Scanner next…* method generates an ***InputMismatchException***
  - Program is terminated; remaining statements are not executed.
- *See Example 6.9 ReadInteger.java*

Solving the Input Problem

- We can check before we read, that the next token matches our expected input.

- The ***Scanner*** class provides *hasNext*… methods for doing this. The *hasNext*… methods return *true* if the next token can be read as the data type specified.

HOME

Scanner Class *hasNext…* Methods

- Each method returns *true* if the next token in the input stream can be read as the data type requested, and *false* otherwise.

Return type	Method name and argument list
`boolean`	`hasNextInt( )`
`boolean`	`hasNextDouble( )`
`boolean`	`hasNextFloat( )`
`boolean`	`hasNextByte( )`
`boolean`	`hasNextShort( )`
`boolean`	`hasNextLong( )`
`boolean`	`hasNextBoolean( )`
`boolean`	`hasNext( )`

HOME

Reprompting for Valid Input

- If the *hasNext* method returns *false*, we need to notify the user that the value typed is not valid and reprompt for new input.

- First we need to flush the invalid input using the *nextLine* method of the *Scanner* class. Then we just ignore that input.

HOME

Scanner nextLine Method

Return type	Method name and argument list
`String`	`nextLine( )` returns the remaining input on the line as a *String*

- Pseudocode for type-safe input:

```
prompt for input
while ( input does not match type requested )
{
    flush input
    reprompt
}
perform read
```

- *See Example 6.10 TypeSafeReadInteger.java*

HOME

Constructing Loop Conditions

- The loop body is executed as long as the loop condition evaluates to *true*

- So if we want to stop executing the loop when the sentinel value is read, the loop condition has to check that the value is NOT the sentinel

- Thus, the **loop continuation condition** is the inverse of the **loop termination condition.**

HOME

Example: Menu Program

- Two sentinel values (*'s'* or *'S'*)
- We are inclined to form this **incorrect** condition:

```
while ( option != 'S'  || option != 's' )
```

- This causes an endless loop because one of the conditions is always *true*

HOME

Constructing a Loop Condition

1. Define the loop termination condition, that is, define the condition that will make the loop stop executing.
2. Create the loop continuation condition – the condition that will keep the loop executing – by applying the Logical NOT operator (!) to the loop termination condition.
3. Simplify the loop continuation condition by applying DeMorgan's Laws, where possible.

HOME

DeMorgan's Laws (see Chapter 5)

- Set of rules to help develop logical expressions that are equivalent

`NOT( A AND B )` is equivalent to

`( NOT A ) OR ( NOT B )`

`NOT( A OR B )` is equivalent to

`( NOT A ) AND ( NOT B )`

HOME

According to DeMorgan's Laws:

```
!( a && b )
```

is equivalent to

```
( !a ) || ( !b )
```

```
!( a || b )
```

is equivalent to

```
!a && !b
```

HOME

Negating Expressions

expression	! (expression)
a = = b	a != b
a != b	a = = b
a < b	a >= b
a >= b	a < b
a > b	a <= b
a <= b	a > b

HOME

The Menu Condition Revisited

1. Define the loop termination condition:

```
( option == 'S' || option == 's' )
```

2. Create the loop continuation condition by applying the **!** operator:

```
! ( option == 'S' || option == 's' )
```

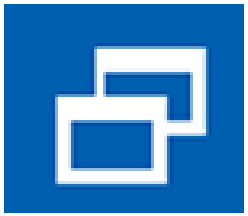
3. Simplify by applying DeMorgan's Laws:

```
( option != 'S' && option != 's' )
```

   This condition is correct!

- *See Example 6.11 CellService.java*

HOME



Do not check for the sentinel value inside a *while* loop. Let the *while* loop condition detect the sentinel value.

Note in Example 6.11 that no code in the loop checks for *'s'* or *'S'*. Because the *while* loop condition does the checking, the *option* variable can never have the value *'s'* or *'S'* inside the loop body. This is true because we use a priming read and an update read.

HOME

A Compound Loop Condition

- Suppose we want to animate the ball so that it rolls diagonally.

- In this case, we will have two possible events:
  - the ball has passed the horizontal border
  - the ball has passed the vertical border

HOME

Develop the Loop Condition

1. Loop termination (ball is out of bounds)

```
( ball.getX( )+ diameter > windowWidth
    || ball.getY( )+ diameter > windowHeight )
```

2. Loop continuation (ball is not out of bounds)

```
! ( ball.getX( )+ diameter > windowWidth
    || ball.getY( )+ diameter > windowHeight )
```

3. Simplify (ball is in bounds)

```
( ball.getX( )+ diameter <= windowWidth
   && ball.getY( )+ diameter <= windowHeight )
```

- *See Example 6.12 RollABall3.java*

HOME

Testing Techniques

1. Does the program produce correct results with a set of known input values?
2. Does the program produce correct results if the sentinel value is the first and only input?
3. Does the program deal appropriately with invalid input?

HOME

Testing Technique 1

1. Does the program produce correct results with known input?

- To verify, select input values, calculate by hand, and compare output to hand-calculated values
- Check boundary values
  - Such as lowest or highest expected values
- Check "edge" values of *if* statements

  For Example, with this *if* condition:

  ```
  ( age >= 18 )
  ```

  we should test the program with values 17, 18, and 19

HOME

Testing Techniques 2 and 3

2. Does the program produce correct results if the sentinel value is the first and only input?

   Result: the *while* loop is not executed; will reported results be correct?

   Test: Enter sentinel value at first prompt

3. Does the program deal appropriately with invalid input?

   Possible results: an Exception is generated or an incorrect action is performed

   Test: Enter invalid data

HOME

The *do/while* Loop

- Unlike the *while* loop, the condition for the *do/while* loop is evaluated at the end of the loop
- Thus, *do/while* loop executes at least once
- Some uses for a *do/while* loop:
  - Validate user input
  - Ask if user wants to repeat an operation

HOME

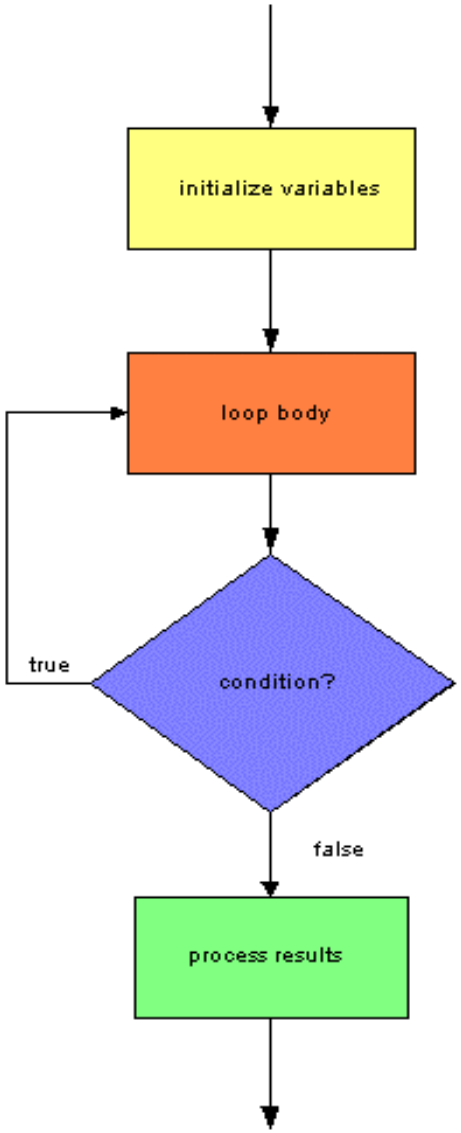
do/while Syntax

```
//initialize variables
do
{
   // body of loop

} while ( condition );
//process the results
```

HOME

do/while Flow of Control



HOME

Example:Validate User Input

- Prompt user inside the *do/while* loop
- Condition is *true* if user entered invalid data, so looping continues until user enters valid data.

- See *Example 6.13 ValidateInput.java*

HOME

- Do not use an *if* statement to validate input because it will catch invalid values entered the first time only.

- A ***do/while*** loop will continue to prompt the user until the user enters a valid value.

HOME

To Repeat an Operation

- Example code to prompt user to play again.

```
do
{
   // code to play a game
   System.out.print( "play again? ")
   String answer = scan.next( );
} while ( answer.equalsIgnoreCase( "yes" ) );
```

HOME

The *for* Loop

- Ideal when you know the number of iterations to perform before the loop begins

- Examples:
  - Find the sum of 5 numbers
  - Find the maximum of 20 numbers
  - Print the odd numbers from 1 to 10

HOME

The *for* Loop Syntax

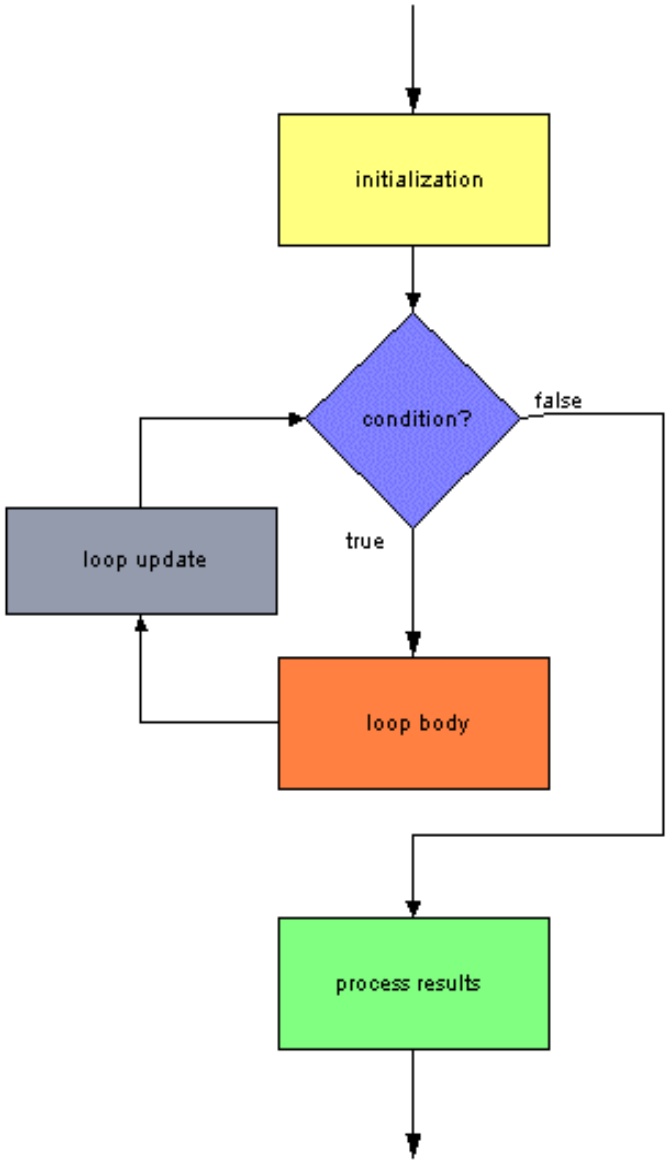
```
for ( initialization; loop condition; loop update )
{
   // loop body
}
```

Notes:

- semicolons separate terms in the loop header
- no semicolon follows the loop header
- curly braces are required only if more than one statement is in the loop body

HOME

for Loop Flow of Control



HOME

for Loop Flow of Control

1. The initialization statement is executed (once only).
2. The loop condition is evaluated. If the condition is *true*, the loop body is executed.
3. The loop update statement is executed, and the loop condition is reevaluated (#2).

- And so on, until the condition is *false*.

HOME

Using a Loop Control Variable

- A **loop control variable** is usually used for counting.
  - We set its initial value in the initialization statement
  - We check its value in the loop condition
  - We increment or decrement its value in the loop update statement

HOME

Example: Find Sum of 5 Integers

```
set total to 0
for i = 1 to 5 by 1
{
    read integer
    add integer to total
}
print the total
```

See Example 6.15 Sum5Numbers.java

HOME

Update Increment Can Be > 1

- Print the even numbers from 0 to 20

```
set output to an empty String
for i = 0 to 20 by 2
{
   append i and a space to output
}
print the output String
```

See Example 6.16 PrintEven.java

HOME

Loop Control Variable Scope

- When a loop control variable is declared inside the *for* loop header, it cannot be referenced after the loop

```
for ( int i = 0; i < 3; i++ )
{
   System.out.println( i ); // ok
}
System.out.println( i ); // error: i undefined
```

HOME

To Reference *i* After the Loop

```
int i;  // declare i before loop
for ( i = 0; i < 3; i++ )
{
  System.out.println( i );
}
System.out.println( i );  // ok
```

HOME

Decrementing the Loop Variable

- Print a string backwards:

```
set backwards to an empty String
read a sentence
for i = ( length of sentence - 1 )
                to 0 by -1
{
   get character at position i
   append character to backwards
}
print backwards
```

- See *Example 6.17 Backwards.java*

HOME

Drawing a Bull's Eye Target

- Draw 10 concentric circles
  - All circles have the same center point
  - Each circle has a different diameter
- To alternate colors (black & red), we use a **toggle variable**
  - variable alternates between two values
- We must draw the circles from the largest to the smallest to avoid covering the smaller circles.

HOME

Pseudocode for Bull's Eye

```
initialize color to black
for diameter = 200 to 20 by -20
{
    instantiate a circle
    draw the circle
    if color is black
        set color to red
    else
        set color to black
}
```

- *See Example 6.18 Bullseye.java*

HOME

Testing *for* Loops

- An important test for *for* loops is that the starting and ending values of the loop variable are set correctly.
- For example, to iterate 5 times, use this header:

```
for ( int i = 0; i < 5; i++ )
```

or this header:

```
for ( int i = 1; i <= 5; i++ )
```

HOME

Processing a *String* (named *word*)

- Forward direction:
  - Correct:

```
for ( int i = 0; i < word.length( ); i++ )
```

  - Incorrect:

```
for ( int i = 0; i <= word.length( ); i++ )
```

- Reverse direction:
  - Correct:

```
for ( int i = word.length( ) - 1; i >= 0; i-- )
```

  - Incorrect:

```
for ( int i = word.length( ); i >= 0; i-- )
```

```
for ( int i = word.length( ) - 1; i > 0; i-- )
```

HOME

Testing *for* Loops

- Test with data that causes *for* loop to execute 0 times (no iterations).

- Example: Test *Example **6.17** Backwards.java* with an empty sentence.

Nested Loops

- Loops can be nested inside other loops; that is, the body of one loop can contain another loop.

- A *while* loop can be nested inside another *while* loop or a *for* loop can be nested inside another *for* loop.

- A *for* loop can be nested inside a *while* loop and a *while* loop can be nested inside a *for* loop.

HOME

Example: Grocery Checkout

```
look for a customer in line
while ( there is a customer in line )
{
    set total to $0.00
    reach for first item
    while item is not the divider bar
    {
        add price to total
        reach for next item
    }
    output the total price
    look for another customer in line
}
```

HOME

Nested *for* Loop Execution

- Inner loop executes all its iterations for each single iteration of the outer loop
- Example: how can we print this?

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

HOME

Analysis

- The highest number we print is the same as the line number.

```
for line = 1 to 5 by 1
{
    for number = 1 to line by 1
    {
        print number and a space
    }
    print a new line
}
```

See Example 6.19 NestedForLoops.java

HOME

Backup

Finding Factors

- We'll let the user enter positive integers, with a 0 being the sentinel value.
- For each number, we'll find all its factors; that is, we will find all the integers that are evenly divisible into the number
- We will not process 1 or the number itself.
- If a number is evenly divisible by another, the remainder after division will be 0. Thus, the modulus operator (%) will be useful.

HOME

Finding Factors (con't)

- To find all the factors of a number, we can test all integers from 1 up to the number, counting all those whose remainder after division is 0.
- But: The number 1 is a factor for every number.
- So we can begin testing at 2.
- And because 2 is the smallest possible factor, there's no need to test integers higher than *number / 2*. Thus, our range of integers to test will be from 2 to *number / 2*.

HOME

Finding Factors Pseudocode

```
read first number  // priming read
while number is not 0
{
   print "The factors for number are "
   for factor = 2 to ( number / 2 ) by 1
   {
      if number % factor is 0
            print factor and a space
   }
   print a new line
   read next number  // update read
}
```

HOME

Finding Factors (con't)

- If no factors are found, the number is prime.
- We need a **flag variable**
  - We set the flag to *false* before starting the *for* loop that checks for factors.
  - Inside the *for* loop, we set the flag to *true* when we find a factor.
  - After the *for* loop terminates, we check the value of the flag. If it is still *false*, we did not find any factors and the number is prime.
- *See Example 6.20 Factors.java*

HOME